Phase II Design for HBASE-10070

04/14/2014

Enis Soztutar (enis@apache.org)
Devaraj Das (ddas@hortonworks.com)

Issue : HBASE-10070 (parent issue)
Doc version : 1.0 (initial 04/14/2014)
Doc version : 1.1 (updated at 10/14/2014)

Related docs

Parent doc

https://docs.google.com/a/hortonworks.com/document/d/1CgEn_b1we5NksR09xuf0PLL3Ea_IC4nBsz-VjunXK8k/edit#heading=h.8xotlio2ebrc

User level guide

https://docs.google.com/a/hortonworks.com/document/d/1N3VWLeIOxz7JqNybusnwOGkrQ1IAhiQlfTKotnZSJIg/edit#heading=h.to8tdc3x6htg

Introduction

This doc will cover changes scheduled for Phase II of HBase-10070, that builds upon Phase I features. We assume Phase 1 features as they are committed in HBASE-10070 branch as of 05/14/2014.

Async WAL Replication

“Async WAL replication feature” as defined in Region Changes section in the parent doc is one of the three proposals for the secondary region replicas to update their statuses with the new data from primary region replica. The three approaches are

- Region Snapshots
- WAL tailing
- Async WAL replication

WAL tailing, and async WAL replication are similar with a major distinction. WAL tailing can be thought as a pull-based data replication scheme, versus Async wal replication is push-based.

Region snapshots

Region snapshots have already been implemented in the HBASE-10070 branch (and merged to branch-1) in issues HBASE-10352 and HBASE-10859. With this feature when we open the secondary region replicas, we do list the store files of the primary, and open the files for reading. We do ensure that we can only do reads in directory, and not disrupt the primary region. This feature is useful for read-only use cases, where after writing the data to the table, we can do a flush, and reopen the table (increase region replication). Then this read-only data will be served by multiple region replicas for higher availability. If the data is read-only, all reads can be done with TIMELINE consistency to have high availability for the data.

The second use case for region snapshots is to enable an automatic file refresher for secondary regions. If enabled, this refresher chore is run in every region server periodically. It does a file list for primary region's files for every region hosted in secondary mode. When the files of the primary region changes (due to flushes and compaction), the file will be picked and opened in the secondary region as well. This means that the new data will only become available after the primary does a flush, then after some time the secondary does the periodic file list and sees the new file. We think that this is useful for bulk-load only tables, and for read-write tables that the user is fine with seeing more stale data.

WAL Tailing

As discussed in the parent doc, we think that WAL tailing only makes sense in a WAL-per-region implementation. Since every secondary region will want to tail the logs of the primary region, without having a separate file for each region, the cluster will end up with all region servers tailing the logs of every other region server, which is not desired. That is why for this work, we will propose the push-based async wal replication feature instead of pull-based wal tailing.

Async WAL Replication (design)

This design builds on the region snapshots and current replication / log replay features.

In issue HBASE-11367, we have made the replication pluggable for this HBASE-10070 work. When a replication peer is added to the cluster, an optional ReplicationEndpoint(RE) class can be passed to the peer definition. RE is an interface that is responsible for shipping the WAL edits to the remote cluster. The ReplicationSource interface and the rest of the WAL

tailing mechanics are kept intact with only the final layer (of shipping the edits) is made pluggable. This enables all implementations to share the same wal-tailing logic, code and admin operations. Given that there is a RE implementation for replicating to a third party, the same add-peer, remove-peer, disable-peer kind of operations, together with replication metrics etc are the same to manage the setup. RE interface does not assume much about the other cluster, enabling replicating to hbase or non-hbase clusters (SOLR, ElasticSearch, MySQL, etc or replicating to another HBase cluster using REST gateway). HBase's own inter-cluster replication is also changed to implement the RE interface giving uniformity. RE also defines a filtering mechanism for WAL entries through the WALEntryFilter interface.

Async WAL replication for HBASE-10070 defines its own RE (see HBASE-11568). As per above, this RE shares the same infrastructure with the regular inter cluster replication, sharing the same wal tailing, zk data structures, and admin operations. This also means that the performance and latency guarantees for async wal replication to replicas similar to the regular replication. As a reminder, the regular replication tails the log for each region server and batches and ships them to the other cluster. When tailing a log that is currently open for writing, it can tail the latest lenght with a delay.

If enabled in configuration, the master adds a peer-id, named "region_replica_replication" whenever a table is created with region replication > 1. This replication peer has the custom RE implementation called RegionReplicaReplicationEndoint (RRRE). Unlike the regular inter cluster replication, which sends the edits to a peer remote cluster, RRRE replicates the WAL edits to the secondary replicas on the same cluster.

If "region_replica_replication" peer is enabled, every region server will host a single ReplicationSource which does the WAL tailing. RRRE only filters out edits from system tables. All other edits are passed (regardless of replication scope). After some batching, it hands over a batch of wal edits to the RRRE. RREE tries to send the edits until it succeeds or is stopped. For this batch, the edits will be grouped by the region name using log replay data structures. For every edit group, it does a meta lookup (cached) to obtain the current region covering these edits, and also obtains all the region replicas. If the region has more than 1 replica, it then sends the edits to all the replicas in parallel through an RPC call. The WAL edits contain the original region name, but at the time of the async wal replication, the regions may have changed via region splits, merges or a table truncate. For these cases, every time a region location is looked up, we check whether this is still the same region replica that we are trying to replay. If not, the edit is skipped. This is a safe operation and the following sections will cover more detail for the reasons.

RRRE uses the replay() rpc introduced by the "distributed log replay" feature. The replay applies the edits to the memstore with the given seqId. Unlike log replay, the replay call on a secondary region replica will not send the edits to the WAL. This is because otherwise, a table with region replication = 3 will persist the edits to WAL 3 times increasing the WAL write amplification.

WAL changes

Actions regarding flushes, compactions, bulk loading as well as region open and close events are persisted to WAL from the primaries. The secondary region replicas never write anything to their WAL.

All of these new types of wal edits are also sent to the secondary region replica via async wal replication. All the events written to WAL contain enough information to replay by the secondary region replica. For flushes, it includes the new file names, for compactions, the new and old file names and store information is included.

Semantics

Edits from the primary are not affected since the secondaries receive the edits from the asynchronous wal replication. This also implies that the secondaries will still have (possibly) stale data, although the staleness will be reduced to a couple of seconds, and will be similar in characteristic to multi-DC semantics and performance.

Memstore and file management

It is critical that the secondaries share the data files with the primary so that the data is not replicated. Only the memstore data is replicated. We also do not want to secondaries to persist the edit to their own WAL files because this will multiply the WAL writes, which will result in significant performance loss on write side.

Memstore management on the secondary

Edits are written to the primary's memstore, and the WAL of the primary region server without any changes. Once this edit is sent over to the secondary, it is put to the memstore of the secondary.

Memstore snapshots already keeps track of the seqId, and saves this in the flushed hfiles. However, the individual cells inside memstore do not keep track of seqIds. This means that the only way to efficiently discard some of the memstore would still be to mimic the memstore snapshots of the primary region in the secondaries. For this, we reintroduced WALEdits for persisting flush actions in WAL. There are four new types of WALEdits, (a)start flush, (b) commit flush, and (c) abort flush and (d) cannot flush. These are sent over to the secondaries. Whenever a secondary sees a start_flush entry, it creates a snapshot of its own memstores by that seqId. Since the edits are already replayed in order, the memstore snapshot contents should be identical to the primary. If the secondary sees the commit flush entry, it will open

the the hfile readers and discard the memstore snapshot. In case of abort, it keeps the memstore snapshot around.

In the secondary region replica side, the wal flush events corresponding to start, abort or commit flush entries will be arriving in order, and each such event will contain the seqId that corresponds to the memstore snapshot seqid in the primary. On regular operation, a flush start will be observed with some seqId, and afterwards some more edits will be replayed. After some time when the flush is finished, a commit flush entry will be observed corresponding to the same seqId.

The primary region might actually be killed or aborted before the ongoing flushes are committed or aborted. This means that the secondaries will have a memstore snapshot that is already abandoned. The primary will be opened in some other region server, and it will execute another flush which may contain more data than the aborted snapshot. Due to different failure conditions, we can observe commit flush without the corresponding start flush, or observe a commit flush with a seqId that is different than what the memstore snapshot has.

In current memstore implementation, there is only one memstore and a at most one snapshot (each is a skip list map). One option we could have pursued is to allow multiple memstore snapshots: This will allow us to keep the memstore snapshot from aborted flush as it is. Whenever a new flush is seen without the previous one being complete, we can create another snapshot of the secondary memstore. The queries will be answered from all memstores which are still ordered by seqId. When a commit flush or abort flush is seen, then we can discard all snapshots with a smaller seqId.

Instead we have implemented a simpler approach. On observing a flush start with some seqId, we create a memstore snapshot with that seqId. If there is already a memstore snapshot that was prepared (with a different seqid) then we do not take an action other thant logging this. On observing a flush commit entry with some seqid, there are multiple possibilities that we can be in:

- A memstore snapshot with the same seqId is prepared corresponding to start flush from before: This is the usual case. We pick up the new flush file and drop the memstore snapshot.
- A memstore snapshot with a smaller seqId is prepared corresponding to start flush from before: This can happen in some failure scenarios. We pick up the new file, we discard the memstore snapshot, and lock and check the max seqId of the edits in the current memstore. If they are less than the flush seqId, we also drop those contents. If not, we keep them around.
- A memstore snapshot with a larger seqId is prepared corresponding to start flush from before: This should not happen normally. However, covered for safety. We pick up the new file, but not drop the memstore or its snapshot.

- No memstore snapshot found from before: In this case, we pick up the new flushed file, and check whether we can safely drop the memstore contents as well using the seqIds.

Compaction events are also picked from replayed WAL entries. Whenever a secondary sees a compaction entry, it applies the compaction, picking new files from compaction and dropping the old ones. Compaction and flushes being written to WAL implies that there is no more need for a store file refresher as defined and implemented in region snapshots section.

Secondaries also replay the bulk load events from the primary since the bulk load action is persisted to WAL. When a secondary sees the marker, it picks up the corresponding bulk load files.

Primary regions write a corresponding region open, or region close event to their wal when the region server performs the action. The region open wal edit contains all the stores and files that the primary opened at the time of the region open event. This wal edits are also replayed in the secondaries. For region close event, the secondaries do not take any action. For region open events, they simply sync their open files to the files that were reported from the event from the primary. Also after this, the secondaries do check their memstores and memstore snapshots to see whether they can drop the edits using the seqId.

Handling memory pressure for secondary region replicas

There have been two proposals for how to handle memory pressure for region servers handling the secondary region replicas. Both of the proposals are included here. However we have chosen to implement the first proposal for it's simplicity. If needed, we will consider going back to second approach as well.

No flushes from secondary regions

We do not allow the secondary region replicas to do flushes by themselves. The edits from the primary is replayed in order to the secondary including flush start and commit markers. In normal operation, the primary will do regular flushes when the memstore limits are achieved. These flushes will also be replayed in secondaries, thus freeing up the memstore memory. It is expected that region servers are hosting primary and secondary regions together in a balanced cluster. In cases where the global memstore limit lower is achieved, the region server looks for candidate memstores to flush and selects the biggest one. For this selection, the secondaries simply do not participate. However, it can happen that all global memstore memory is filled with mostly secondary region replicas resulting in no flushes to clean up space. For preventing indefinite blocking due to flush events not being replayable (because of lack of memstore space), we allow the secondary regions to do a refresh store files operation similar to the one defined in "region snapshots" section. This operation will do an ls for the

primary regions store files, and picks up missing store files. After this, if there are new files with higher seqIds, the memstore is inspected to see whether we can drop the edits because of all the seqId's are smaller. If we are able to drop the memstore contents, it means the replay is unblocked and we can continue. One drawback for this approach is that, since every column family do flushes independently, the atomic row updates may be seen partially from secondaries when a secondary picks up the flush file for a particular cf before the others.

Allow Flushes for the secondary regions

The approach here is to instead allow the secondaries to "spill-to-disk" (in this case it is a simple flush). That is, the regionserver considers secondaries like any other region for flush candidates. However, the spill is done to a configured place on the local disk since these files are really temporary and even if the machine dies or something, the data is not lost since the data is still there on the primary side (in storefiles and in WALs). In either of the approaches, new secondaries can be brought up to speed based on that data. The secondary serves reads from a merged view of primary's store files, secondary's memstore and secondary's spill file (which would be yet another storefile).

Periodically, or on certain events, the spills should be cleaned up. There are multiple approaches here, for example, periodic scan of the files and discard them based on the seqId persisted (just like regionserver handles cleanup of WAL files), or, when we see a flush-commit message from the primary (if we reintroduce the flush-commit message as discussed in the previous approach), we archive/discard some of those spilled-files (that's a logical point since the secondary updates its view of the store files then). We feel that there won't be that many spills accumulated if we follow a simple cleaning up process. This approach has associated writes to local disk, which is something that's not needed in the previous approach. But it doesn't have any (or little, if any) state management and co-ordination to do with flushes (client throttling etc) on the primary side as discussed in the previous approach, and flows pretty well with the normal regionserver operations (very less special casing of memstore handling etc).

Failure handling (Primary and Secondaries)

In case of the RS hosting the primary region going down, the replication queue already handles failover, and ensures that recoveredLease() is called to see all the data. Though as already reported elsewhere, region moves and failures can cause concurrent or out-of-order delivery of wal edits to the receiving side. However, with the addition of writing region open events to WAL, we cover this out-of-order delivery issues. If a region is moved or rs failover, the new region server will first open the region and write a region open marker with a seqId. This seqId is monotonically increasing. Since this is the first item in a log for a region, the replaying side takes advantage of this to reason about the order. The secondary region always keeps a "largest seen region open seqId" variable. This gets updated everytime a new region open event is seen. For every replay of WAL edit, we check the seqId of the edit with this number and silently ignore (not apply) the edit if its seqId is smaller. This allows the

secondary region to not replay events from older wal replays. Since the region open also contains all the store files at the time of primary region opening, we guarantee that ignoring previous entries are safe (since we should have already picked up those edits through a flush).

In case of RS hosting a secondary region going down, the replication queue for this secondary cannot continue from where it is left because the edits are not persisted at the secondary through WAL (we skip WAL at replay). However, the secondary region will be opened elsewhere, and the store files opened will contain the latest seqId that is persisted. From this point on, replicated edits will start arriving, however the secondary will not start serving data (throwing a special exception) until it sees a next flush commit from the primary. The reason is that, we do not want the secondary to go back in time due to lost WAL entries that is not persisted in its own WAL. Whenever a secondary starts serving, it triggers a flush from the primary region which should write a new file, or if there is nothing to flush, mark an entry in the WAL. Upon seeing a full flush cycle (from start flush to commit flush) or a region open event marker, the secondary starts serving data.

In terms of the WAL file lifecycle, this schema will work with the offset based tracking in replication queue as well. If secondaries lose the edits because of a failover, they will continue to wait until the next sync point (flush) to start serving data.

Region Splits and Merges

Phase 1 requires to use the `DisabledRegionSplitPolicy`.

While the split is executed, the region is closed for a short while, and the daughters reopened at the same region server. A meta multi-mutation decides on the fate of the split. If that is successful, it is only rolled forward. Once the split completes successfully, the master takes the necessary actions on the replicas. It closes the primary and opens replicas for the daughters. Merges of regions is done in a similar fashion.

Replicas for Meta

Replicas for the meta regions needs to be handled differently because of the fact that the location of the meta region is noted in ZooKeeper. The replica locations are also noted in ZooKeeper. The secondary replica locations are not used by the master/regionserver at all (it’s not safe for the master/regionserver to read potentially stale entries).

Performance

We will continue on the performance analysis, and ensure that there is no regression and the overhead of extra RPCs are worth the tradeoff for the use cases.

Other Changes

API-wise, the changes already there in the branch should be enough to cover the use cases. There might be small usability improvements.

Testing, stabilization etc work will continue as expected.

Promotion of secondaries will be detailed later. Since this is a MTTR improvement, and requires WAL replication, we can do this after Phase 2 work is complete.

Future work

Warming Block caches of secondaries

This can be implemented as an optional feature iff testing proves that this is a requirement. In case of the primary region server handling a large percentile of the request without latency issues, the block caches of the secondaries will not be hot with the data. In case of a failover, then the secondaries will be getting a lot of request. With a cold cache, the latency spikes might be intolerable for the application.

For this we can optionally send a small percentage of the requests to the secondaries immediately to pre-warm the block caches and keep the blocks mostly cached. For these requests, the primary response will still be waited until the primaryTimeout has passed so that it will be transparent.